BOOK 5: ARRAYS
Chapter 1: Creating Arrays

WHAT IS AN ARRAY

An array is a list of values stored together:

```
let fruits = ["apple", "banana", "cherry"]
```

Instead of three separate variables:

```
let fruit1 = "apple"
let fruit2 = "banana"
let fruit3 = "cherry"
```

One variable holds all three values. Arrays let you work with collections of related data.

CREATING ARRAYS

Array literal (most common):

```
let numbers = [1, 2, 3, 4, 5]
let names = ["Alice", "Bob", "Charlie"]
let mixed = [1, "two", true, null]
```

Square brackets contain the elements, separated by commas.

Empty array:

```
let emptyList = []
```

You can add elements later.

Array constructor (less common):

```
let numbers = new Array(1, 2, 3)
let sized = new Array(5)
```

The constructor can be confusing. Stick to array literals.

Array.of (creates array from arguments):

```
let numbers = Array.of(1, 2, 3)
```

Array.from (creates array from array-like objects):

```
let letters = Array.from("hello")

console.log(letters)
```

Output: ["h", "e", "l", "l", "o"]

ARRAY ELEMENTS CAN BE ANY TYPE

Strings:

```
let colors = ["red", "green", "blue"]
```

Numbers:

```
let scores = [85, 92, 78, 90]
```

Booleans:

```
let answers = [true, false, true, true]
```

Mixed types:

```
let mixed = [42, "hello", true, null, undefined]
```

Objects:

```
let users = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 }
]
```

Arrays (nested):

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]
```

Functions:

```
let operations = [
 function(a, b) { return a + b },
 function(a, b) { return a - b }
]
```

ARRAY LENGTH

Every array has a length property:

```
let fruits = ["apple", "banana", "cherry"]

console.log(fruits.length)
```

Output: 3

Empty array has length 0:

```
let empty = []

console.log(empty.length)
```

Output: 0

Length updates automatically:

```
let list = []
console.log(list.length)

list.push("item")
console.log(list.length)

list.push("another")
console.log(list.length)
```

Output:
0
1
2

ACCESSING ELEMENTS

Use square brackets with an index:

```
let fruits = ["apple", "banana", "cherry"]

console.log(fruits[0])
console.log(fruits[1])
console.log(fruits[2])
```

Output:
apple
banana
cherry

Indices start at 0, not 1. The first element is at index 0.

Access the last element:

let last = fruits[fruits.length - 1]

console.log(last)

Output: cherry

If length is 3, the last index is 2 (length - 1).

Accessing out of bounds:

console.log(fruits[10])

Output: undefined

No error, just undefined.

MODIFYING ELEMENTS

Assign to an index:

let fruits = ["apple", "banana", "cherry"]

fruits[1] = "blueberry"

console.log(fruits)

Output: ["apple", "blueberry", "cherry"]

The second element (index 1) changed from "banana" to "blueberry".

Add element beyond current length:

let numbers = [1, 2, 3]

numbers[5] = 6

```
console.log(numbers)
console.log(numbers.length)
```

Output:
[1, 2, 3, undefined, undefined, 6]
6

Indices 3 and 4 are empty slots (undefined).

CHECKING IF ARRAY

Use Array.isArray:

```
let arr = [1, 2, 3]
let notArr = "hello"

console.log(Array.isArray(arr))
console.log(Array.isArray(notArr))
```

Output:
true
false

Do not use typeof:

```
console.log(typeof arr)
```

Output: "object"

Arrays are objects, so typeof returns "object", not "array".

PRACTICAL EXAMPLES

Shopping list:

```
let shoppingList = ["milk", "bread", "eggs", "butter"]

console.log(`Items to buy: ${shoppingList.length}`)
console.log(`First item: ${shoppingList[0]}`)
console.log(`Last item: ${shoppingList[shoppingList.length - 1]}`)
```

Student scores:

```
let scores = [85, 92, 78, 90, 88]

console.log(`Scores: ${scores}`)
console.log(`First score: ${scores[0]}`)
console.log(`Number of scores: ${scores.length}`)
```

Configuration:

```
let enabledFeatures = ["dark-mode", "notifications", "auto-save"]

console.log(`Features enabled: ${enabledFeatures.length}`)
```

Coordinates:

```
let point = [10, 20]
let x = point[0]
let y = point[1]

console.log(`Position: (${x}, ${y})`)
```

RGB color:

```
let red = [255, 0, 0]
let green = [0, 255, 0]
let blue = [0, 0, 255]

console.log(`Red: rgb(${red.join(", ")})`)
```

ARRAYS VS OBJECTS

Arrays are ordered lists with numeric indices:

```
let fruits = ["apple", "banana", "cherry"]
console.log(fruits[0])
```

Objects have named properties:

```
let person = { name: "Alice", age: 25 }
console.log(person.name)
```

Use arrays when:
- Order matters
- You have a list of similar items
- You need to loop through items

Use objects when:
- Items have names or labels
- You need to access by key
- Items represent properties of something

PRACTICE

Create an array of your five favorite foods.

Access the first and last elements of the array.

Change the third element to something else.

Create an empty array and check its length.

Create a nested array (array of arrays).

WHAT YOU LEARNED

You learned that arrays hold multiple values in order.
You learned to create arrays with square brackets.
You learned that elements can be any type.
You learned that indices start at 0.
You learned to access elements with bracket notation.
You learned to modify elements by assigning to an index.
You learned to check array length with .length.

The next chapter teaches you how to add and remove elements.

BOOK 5: ARRAYS
Chapter 2: Adding and Removing Elements

ADDING TO THE END

The push method adds elements to the end:

```
let fruits = ["apple", "banana"]

fruits.push("cherry")

console.log(fruits)
```

Output: ["apple", "banana", "cherry"]

Push returns the new length:

```
let length = fruits.push("date")

console.log(length)
console.log(fruits)
```

Output:
4
["apple", "banana", "cherry", "date"]

Add multiple elements:

fruits.push("elderberry", "fig")

console.log(fruits)

Output: ["apple", "banana", "cherry", "date", "elderberry", "fig"]

REMOVING FROM THE END

The pop method removes the last element:

let fruits = ["apple", "banana", "cherry"]

let removed = fruits.pop()

console.log(removed)
console.log(fruits)

Output:
cherry
["apple", "banana"]

Pop returns the removed element.

Pop from empty array:

let empty = []

console.log(empty.pop())

Output: undefined

ADDING TO THE BEGINNING

The unshift method adds elements to the beginning:

```
let fruits = ["banana", "cherry"]

fruits.unshift("apple")

console.log(fruits)
```

Output: ["apple", "banana", "cherry"]

Add multiple:

```
fruits.unshift("aardvark", "apricot")

console.log(fruits)
```

Unshift returns the new length.

REMOVING FROM THE BEGINNING

The shift method removes the first element:

```
let fruits = ["apple", "banana", "cherry"]

let removed = fruits.shift()

console.log(removed)
console.log(fruits)
```

Output:
apple
["banana", "cherry"]

Shift returns the removed element.

PUSH/POP VS UNSHIFT/SHIFT

push: add to end
pop: remove from end
unshift: add to beginning
shift: remove from beginning

End operations (push/pop) are faster than beginning operations (unshift/shift). Adding or removing from the beginning requires shifting all other elements.

SPLICE: THE POWERFUL METHOD

The splice method can add, remove, or replace elements anywhere:

splice(startIndex, deleteCount, item1, item2, ...)

Remove elements:

```
let letters = ["a", "b", "c", "d", "e"]

let removed = letters.splice(1, 2)

console.log(removed)
console.log(letters)
```

Output:
["b", "c"]
["a", "d", "e"]

Starting at index 1, remove 2 elements.

Insert elements (delete 0):

let letters = ["a", "d", "e"]

letters.splice(1, 0, "b", "c")

console.log(letters)

Output: ["a", "b", "c", "d", "e"]

At index 1, delete 0 elements, insert "b" and "c".

Replace elements:

let letters = ["a", "x", "x", "d", "e"]

letters.splice(1, 2, "b", "c")

console.log(letters)

Output: ["a", "b", "c", "d", "e"]

At index 1, delete 2 elements, insert "b" and "c".

SLICE: COPY A PORTION

The slice method creates a new array from part of the original:

slice(startIndex, endIndex)

let fruits = ["apple", "banana", "cherry", "date", "elderberry"]

let some = fruits.slice(1, 4)

console.log(some)
console.log(fruits)

Output:
["banana", "cherry", "date"]

["apple", "banana", "cherry", "date", "elderberry"]

Slice does not modify the original. It returns a new array.

Start to end:

let first3 = fruits.slice(0, 3)

console.log(first3)

Output: ["apple", "banana", "cherry"]

From index to end:

let last3 = fruits.slice(2)

console.log(last3)

Output: ["cherry", "date", "elderberry"]

Negative indices count from the end:

let lastTwo = fruits.slice(-2)

console.log(lastTwo)

Output: ["date", "elderberry"]

Copy entire array:

let copy = fruits.slice()

console.log(copy)

CONCAT: COMBINE ARRAYS

The concat method joins arrays:

```
let arr1 = [1, 2, 3]
let arr2 = [4, 5, 6]

let combined = arr1.concat(arr2)

console.log(combined)
console.log(arr1)
console.log(arr2)
```

Output:
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
[4, 5, 6]

Concat does not modify originals. It returns a new array.

Multiple arrays:

```
let result = arr1.concat(arr2, [7, 8], [9, 10])

console.log(result)
```

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

SPREAD OPERATOR

The spread operator (...) expands an array:

```
let arr1 = [1, 2, 3]
let arr2 = [4, 5, 6]

let combined = [...arr1, ...arr2]

console.log(combined)
```

Output: [1, 2, 3, 4, 5, 6]

Add elements with spread:

let withExtra = [...arr1, 99, ...arr2]

console.log(withExtra)

Output: [1, 2, 3, 99, 4, 5, 6]

Copy array:

let copy = [...arr1]

Spread is often cleaner than concat.

PRACTICAL EXAMPLES

Todo list:

```
let todos = []

function addTodo(task) {
 todos.push(task)
 console.log(`Added: ${task}`)
}

function completeTodo() {
 let done = todos.shift()
 console.log(`Completed: ${done}`)
}

addTodo("Buy groceries")
addTodo("Walk the dog")
addTodo("Do laundry")

console.log(todos)
```

```
completeTodo()

console.log(todos)
```

Stack (last in, first out):

```
let stack = []

stack.push("first")
stack.push("second")
stack.push("third")

console.log(stack.pop())
console.log(stack.pop())
console.log(stack.pop())
```

Output:
third
second
first

Queue (first in, first out):

```
let queue = []

queue.push("first")
queue.push("second")
queue.push("third")

console.log(queue.shift())
console.log(queue.shift())
console.log(queue.shift())
```

Output:
first
second
third

Remove specific item:

```
function removeItem(array, item) {
 let index = array.indexOf(item)
 if (index !== -1) {
 array.splice(index, 1)
 }
}

let list = ["a", "b", "c", "d"]
removeItem(list, "c")

console.log(list)
```

Output: ["a", "b", "d"]

Insert in order:

```
function insertSorted(array, value) {
 let i = 0
 while (i < array.length && array[i] < value) {
 i++
 }
 array.splice(i, 0, value)
}

let sorted = [1, 3, 5, 7]
insertSorted(sorted, 4)

console.log(sorted)
```

Output: [1, 3, 4, 5, 7]

MUTATING VS NON-MUTATING

Methods that change the original array (mutating):
- push, pop, shift, unshift
- splice
- sort, reverse

Methods that return a new array (non-mutating):
- slice
- concat
- map, filter (covered later)

Know which methods modify and which copy. This prevents unexpected changes.

PRACTICE

Create an array and add items with push.

Remove the last item with pop and the first with shift.

Use splice to insert an item in the middle.

Use slice to copy part of an array.

Combine two arrays with concat or spread.

WHAT YOU LEARNED

You learned push adds to the end, pop removes from the end.
You learned unshift adds to the beginning, shift removes from the beginning.
You learned splice can insert, remove, or replace anywhere.
You learned slice copies a portion without modifying.
You learned concat and spread combine arrays.
You learned which methods mutate and which create new arrays.

The next chapter teaches you how to search arrays.
BOOK 5: ARRAYS
Chapter 3: Searching Arrays

FINDING AN ELEMENT'S INDEX

The indexOf method returns the index of an element:

```
let fruits = ["apple", "banana", "cherry", "banana"]

console.log(fruits.indexOf("banana"))
console.log(fruits.indexOf("cherry"))
console.log(fruits.indexOf("grape"))
```

Output:
1
2
-1

If found, returns the index. If not found, returns -1.

indexOf finds the first occurrence:

```
let numbers = [1, 2, 3, 2, 1]

console.log(numbers.indexOf(2))
```

Output: 1

The first 2 is at index 1.

FINDING LAST OCCURRENCE

The lastIndexOf method searches from the end:

```
let fruits = ["apple", "banana", "cherry", "banana"]
```

```
console.log(fruits.lastIndexOf("banana"))
```

Output: 3

The last "banana" is at index 3.

CHECKING IF ELEMENT EXISTS

The includes method returns true or false:

```
let fruits = ["apple", "banana", "cherry"]

console.log(fruits.includes("banana"))
console.log(fruits.includes("grape"))
```

Output:
true
false

Easier to read than checking indexOf !== -1:

```
if (fruits.includes("banana")) {
 console.log("We have bananas!")
}
```

FINDING WITH A CONDITION

The find method returns the first element that passes a test:

```
let numbers = [1, 5, 10, 15, 20]

let found = numbers.find(num => num > 8)
```

console.log(found)

Output: 10

Find returns the element itself, not its index.

If nothing matches, returns undefined:

let found = numbers.find(num => num > 100)

console.log(found)

Output: undefined

FINDING INDEX WITH A CONDITION

The findIndex method returns the index of the first match:

let numbers = [1, 5, 10, 15, 20]

let index = numbers.findIndex(num => num > 8)

console.log(index)

Output: 2

The element 10 is at index 2.

If nothing matches, returns -1:

let index = numbers.findIndex(num => num > 100)

console.log(index)

Output: -1

FINDING ALL MATCHES

The filter method returns all matching elements:

```
let numbers = [1, 5, 10, 15, 20]

let bigNumbers = numbers.filter(num => num > 8)

console.log(bigNumbers)
```

Output: [10, 15, 20]

Filter returns a new array with all matches.

No matches returns empty array:

```
let huge = numbers.filter(num => num > 100)

console.log(huge)
```

Output: []

CHECKING IF ANY MATCH

The some method returns true if at least one element passes:

```
let numbers = [1, 5, 10, 15, 20]

console.log(numbers.some(num => num > 15))
console.log(numbers.some(num => num > 100))
```

Output:
true
false

At least one number is greater than 15. No number is greater than 100.

CHECKING IF ALL MATCH

The every method returns true if all elements pass:

```
let numbers = [2, 4, 6, 8, 10]

console.log(numbers.every(num => num % 2 === 0))
console.log(numbers.every(num => num > 5))
```

Output:
true
false

All are even. Not all are greater than 5.

PRACTICAL EXAMPLES

Search for user:

```
let users = [
 { id: 1, name: "Alice" },
 { id: 2, name: "Bob" },
 { id: 3, name: "Charlie" }
]

let user = users.find(u => u.id === 2)

console.log(user)
```

Output: { id: 2, name: "Bob" }

Check if product in stock:

```
let inventory = ["laptop", "phone", "tablet", "headphones"]

function isInStock(product) {
 return inventory.includes(product.toLowerCase())
}

console.log(isInStock("Phone"))
console.log(isInStock("Camera"))
```

Output:
true
false

Find all adults:

```
let people = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 17 },
 { name: "Charlie", age: 35 },
 { name: "Diana", age: 16 }
]

let adults = people.filter(person => person.age >= 18)

console.log(adults.map(p => p.name))
```

Output: ["Alice", "Charlie"]

Validate form fields:

```
let fields = [
 { name: "email", value: "user@example.com" },
 { name: "password", value: "" },
 { name: "username", value: "user123" }
]

let hasEmpty = fields.some(field => field.value === "")
```

```
console.log(`Has empty fields: ${hasEmpty}`)
```

Output: Has empty fields: true

Check all scores passing:

```
let scores = [75, 82, 90, 68, 85]

let allPassing = scores.every(score => score >= 60)

console.log(`All passing: ${allPassing}`)
```

Output: All passing: true

Remove item if exists:

```
function removeIfExists(array, item) {
 let index = array.indexOf(item)
 if (index !== -1) {
 array.splice(index, 1)
 return true
 }
 return false
}

let list = ["a", "b", "c"]

console.log(removeIfExists(list, "b"))
console.log(removeIfExists(list, "x"))
console.log(list)
```

Output:
true
false
["a", "c"]

SEARCH WITH START INDEX

indexOf and includes accept a start index:

```
let letters = ["a", "b", "c", "a", "b", "c"]

console.log(letters.indexOf("a", 2))
```

Output: 3

Start searching from index 2.

```
console.log(letters.includes("a", 4))
```

Output: false

No "a" at index 4 or later.

CASE SENSITIVITY

Searches are case-sensitive:

```
let names = ["Alice", "Bob", "Charlie"]

console.log(names.includes("alice"))
console.log(names.includes("Alice"))
```

Output:
false
true

For case-insensitive search:

```
function includesIgnoreCase(array, value) {
 let lowerValue = value.toLowerCase()
 return array.some(item => item.toLowerCase() ===
lowerValue)
}
```

```
console.log(includesIgnoreCase(names, "alice"))
```

Output: true

FIND DUPLICATES

Find first duplicate:

```
function findFirstDuplicate(array) {
 for (let i = 0; i < array.length; i++) {
 if (array.indexOf(array[i]) !== i) {
 return array[i]
 }
 }
 return null
}

let letters = ["a", "b", "c", "b", "d"]

console.log(findFirstDuplicate(letters))
```

Output: b

Find all duplicates:

```
function findAllDuplicates(array) {
 let seen = []
 let duplicates = []

 for (let item of array) {
 if (seen.includes(item) && !duplicates.includes(item)) {
 duplicates.push(item)
 } else {
 seen.push(item)
 }
 }
```

```
 return duplicates
}

let letters = ["a", "b", "c", "b", "d", "a", "e", "a"]

console.log(findAllDuplicates(letters))
```

Output: ["b", "a"]

PRACTICE

Find the index of a specific element in an array.

Check if an array contains a specific value.

Find the first number greater than 50 in an array.

Filter an array to get only strings longer than 5 characters.

Check if all elements in an array are positive.

WHAT YOU LEARNED

You learned indexOf finds the first occurrence.
You learned includes checks existence.
You learned find returns the first matching element.
You learned findIndex returns the index of the first match.
You learned filter returns all matching elements.
You learned some checks if any element matches.
You learned every checks if all elements match.

The next chapter teaches you how to transform arrays.

BOOK 5: ARRAYS

Chapter 4: Transforming Arrays

MAP: TRANSFORM EACH ELEMENT

The map method creates a new array by transforming each element:

```
let numbers = [1, 2, 3, 4, 5]

let doubled = numbers.map(num => num * 2)

console.log(doubled)
console.log(numbers)
```

Output:

```
[2, 4, 6, 8, 10]
[1, 2, 3, 4, 5]
```

Map does not change the original. It returns a new array.

Transform strings:

```
let names = ["alice", "bob", "charlie"]

let upper = names.map(name => name.toUpperCase())

console.log(upper)
```

Output: ["ALICE", "BOB", "CHARLIE"]

Extract properties from objects:

```
let users = [
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 },
 { name: "Charlie", age: 35 }
]
```

```
let names = users.map(user => user.name)

console.log(names)
```

Output: ["Alice", "Bob", "Charlie"]

Transform objects:

```
let products = [
 { name: "Laptop", price: 1000 },
 { name: "Phone", price: 500 }
]

let withTax = products.map(product => ({
 ...product,
 priceWithTax: product.price * 1.08
}))

console.log(withTax)
```

FILTER: KEEP MATCHING ELEMENTS

The filter method creates a new array with elements that pass a test:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = numbers.filter(num => num % 2 === 0)

console.log(evens)
```

Output: [2, 4, 6, 8, 10]

Filter active users:

```
let users = [
 { name: "Alice", active: true },
```

```
 { name: "Bob", active: false },
 { name: "Charlie", active: true }
]

let activeUsers = users.filter(user => user.active)

console.log(activeUsers.map(u => u.name))
```

Output: ["Alice", "Charlie"]

Remove empty strings:

```
let words = ["hello", "", "world", "", "test"]

let nonEmpty = words.filter(word => word !== "")

console.log(nonEmpty)
```

Output: ["hello", "world", "test"]

Filter with truthy check:

```
let cleaned = words.filter(word => word)
```

Same result. Empty strings are falsy.

REDUCE: COMBINE INTO ONE VALUE

The reduce method combines all elements into a single value:

```
let numbers = [1, 2, 3, 4, 5]

let sum = numbers.reduce((accumulator, current) => {
 return accumulator + current
}, 0)

console.log(sum)
```

Output: 15

Parameters:
- accumulator: the running result
- current: the current element
- 0: the initial value

Step by step:
- Start: accumulator = 0
- Step 1: 0 + 1 = 1
- Step 2: 1 + 2 = 3
- Step 3: 3 + 3 = 6
- Step 4: 6 + 4 = 10
- Step 5: 10 + 5 = 15

Product of all:

```
let product = numbers.reduce((acc, cur) => acc * cur, 1)

console.log(product)
```

Output: 120 (1*2*3*4*5)

Find maximum:

```
let numbers = [5, 2, 8, 1, 9, 3]

let max = numbers.reduce((acc, cur) => cur > acc ? cur : acc)

console.log(max)
```

Output: 9

Count occurrences:

```
let letters = ["a", "b", "a", "c", "a", "b"]

let counts = letters.reduce((acc, letter) => {
 acc[letter] = (acc[letter] || 0) + 1
```

```
 return acc
}, {})

console.log(counts)
```

Output: { a: 3, b: 2, c: 1 }

Group by property:

```
let people = [
 { name: "Alice", department: "Engineering" },
 { name: "Bob", department: "Sales" },
 { name: "Charlie", department: "Engineering" }
]

let byDepartment = people.reduce((acc, person) => {
 let dept = person.department
 if (!acc[dept]) {
 acc[dept] = []
 }
 acc[dept].push(person.name)
 return acc
}, {})

console.log(byDepartment)
```

Output: { Engineering: ["Alice", "Charlie"], Sales: ["Bob"] }

SORT: ORDER ELEMENTS

The sort method orders elements in place:

```
let fruits = ["cherry", "apple", "banana"]

fruits.sort()

console.log(fruits)
```

Output: ["apple", "banana", "cherry"]

Default sort is alphabetical by string conversion:

let numbers = [10, 5, 8, 1, 7]

numbers.sort()

console.log(numbers)

Output: [1, 10, 5, 7, 8]

Wrong! 10 comes before 5 because "10" < "5" as strings.

Numeric sort with compare function:

numbers.sort((a, b) => a - b)

console.log(numbers)

Output: [1, 5, 7, 8, 10]

If (a - b) is negative, a comes first.
If (a - b) is positive, b comes first.
If (a - b) is zero, order unchanged.

Descending order:

numbers.sort((a, b) => b - a)

console.log(numbers)

Output: [10, 8, 7, 5, 1]

Sort objects by property:

let users = [
 { name: "Charlie", age: 35 },
 { name: "Alice", age: 25 },

```
 { name: "Bob", age: 30 }
]

users.sort((a, b) => a.age - b.age)

console.log(users.map(u => u.name))
```

Output: ["Alice", "Bob", "Charlie"]

REVERSE: FLIP ORDER

The reverse method reverses in place:

```
let numbers = [1, 2, 3, 4, 5]

numbers.reverse()

console.log(numbers)
```

Output: [5, 4, 3, 2, 1]

FLAT: FLATTEN NESTED ARRAYS

The flat method flattens nested arrays:

```
let nested = [1, [2, 3], [4, [5, 6]]]

console.log(nested.flat())
console.log(nested.flat(2))
```

Output:
[1, 2, 3, 4, [5, 6]]
[1, 2, 3, 4, 5, 6]

flat(1) is default, one level. flat(2) goes two levels deep.

flat(Infinity) flattens all levels:

let deep = [1, [2, [3, [4, [5]]]]]

console.log(deep.flat(Infinity))

Output: [1, 2, 3, 4, 5]

FLATMAP: MAP THEN FLATTEN

The flatMap method maps then flattens one level:

let sentences = ["Hello world", "How are you"]

let words = sentences.flatMap(s = > s.split(" "))

console.log(words)

Output: ["Hello", "world", "How", "are", "you"]

Equivalent to:

let words = sentences.map(s = > s.split(" ")).flat()

CHAINING TRANSFORMATIONS

Chain methods together:

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let result = numbers
 .filter(n = > n % 2 = = = 0)

```
 .map(n => n * n)
 .reduce((sum, n) => sum + n, 0)

console.log(result)
```

Output: 220

Steps:
1. Filter evens: [2, 4, 6, 8, 10]
2. Square each: [4, 16, 36, 64, 100]
3. Sum: 220

Process data pipeline:

```
let users = [
 { name: "Alice", age: 25, active: true },
 { name: "Bob", age: 17, active: true },
 { name: "Charlie", age: 30, active: false },
 { name: "Diana", age: 22, active: true }
]

let result = users
 .filter(u => u.active)
 .filter(u => u.age >= 18)
 .map(u => u.name.toUpperCase())
 .sort()

console.log(result)
```

Output: ["ALICE", "DIANA"]

PRACTICE

Use map to double all numbers in an array.

Use filter to get only positive numbers.

Use reduce to find the sum of an array.

Sort an array of objects by a numeric property.

Chain filter, map, and reduce together.

WHAT YOU LEARNED

You learned map transforms each element into a new array.
You learned filter keeps elements that pass a test.
You learned reduce combines all elements into one value.
You learned sort orders elements with a compare function.
You learned reverse flips the order.
You learned flat flattens nested arrays.
You learned to chain transformations together.

The next chapter teaches you about sorting in depth.

Chapter 5: Sorting in Depth

DEFAULT SORT BEHAVIOR

The default sort converts elements to strings and sorts alphabetically:

```
let letters = ["c", "a", "b"]

letters.sort()

console.log(letters)
```

Output: ["a", "b", "c"]

This works for strings. But numbers have problems:

let numbers = [100, 25, 10, 5, 50]

numbers.sort()

console.log(numbers)

Output: [10, 100, 25, 5, 50]

"10" comes before "25" alphabetically because "1" < "2".

THE COMPARE FUNCTION

Sort takes a compare function that defines the order:

array.sort((a, b) => compareValue)

The compare function receives two elements and returns:
- Negative number: a comes before b
- Positive number: b comes before a
- Zero: keep original order

NUMERIC SORTING

Ascending (smallest to largest):

let numbers = [100, 25, 10, 5, 50]

numbers.sort((a, b) => a - b)

console.log(numbers)

Output: [5, 10, 25, 50, 100]

If a is smaller than b, a - b is negative, so a comes first.

Descending (largest to smallest):

numbers.sort((a, b) => b - a)

console.log(numbers)

Output: [100, 50, 25, 10, 5]

STRING SORTING

Alphabetical (default):

let names = ["Charlie", "Alice", "Bob"]

names.sort()

console.log(names)

Output: ["Alice", "Bob", "Charlie"]

Case-insensitive:

let mixed = ["banana", "Apple", "cherry"]

mixed.sort()

console.log(mixed)

Output: ["Apple", "banana", "cherry"]

Uppercase letters come before lowercase by default.

mixed.sort((a, b) =>
a.toLowerCase().localeCompare(b.toLowerCase()))

console.log(mixed)

Output: ["Apple", "banana", "cherry"]

Using localeCompare for proper string comparison:

mixed.sort((a, b) => a.localeCompare(b, undefined, { sensitivity: "base" }))

The sensitivity option ignores case and accents.

SORTING OBJECTS

Sort by string property:

```
let users = [
 { name: "Charlie", age: 35 },
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 }
]
```

users.sort((a, b) => a.name.localeCompare(b.name))

console.log(users.map(u => u.name))

Output: ["Alice", "Bob", "Charlie"]

Sort by number property:

users.sort((a, b) => a.age - b.age)

console.log(users.map(u => `${u.name}: ${u.age}`))

Output: ["Alice: 25", "Bob: 30", "Charlie: 35"]

Descending by age:

```
users.sort((a, b) => b.age - a.age)

console.log(users.map(u => `${u.name}: ${u.age}`))
```

Output: ["Charlie: 35", "Bob: 30", "Alice: 25"]

MULTI-LEVEL SORTING

Sort by one property, then by another:

```
let products = [
 { category: "Electronics", name: "Phone", price: 500 },
 { category: "Electronics", name: "Laptop", price: 1000 },
 { category: "Clothing", name: "Shirt", price: 30 },
 { category: "Electronics", name: "Tablet", price: 400 },
 { category: "Clothing", name: "Pants", price: 50 }
]

products.sort((a, b) => {
 let categoryCompare = a.category.localeCompare(b.category)
 if (categoryCompare !== 0) {
 return categoryCompare
 }
 return a.price - b.price
})

console.log(products.map(p => `${p.category}: ${p.name}`))
```

Output:
Clothing: Shirt
Clothing: Pants
Electronics: Tablet
Electronics: Phone
Electronics: Laptop

First sorted by category, then by price within each category.

SORT STABILITY

Modern JavaScript engines have stable sort. Elements with equal sort values keep their original relative order.

```
let data = [
 { name: "Alice", score: 90 },
 { name: "Bob", score: 85 },
 { name: "Charlie", score: 90 },
 { name: "Diana", score: 85 }
]

data.sort((a, b) => b.score - a.score)

console.log(data.map(d => d.name))
```

Output: ["Alice", "Charlie", "Bob", "Diana"]

Alice and Charlie both have 90. Alice was first originally, stays first. Bob and Diana both have 85. Bob was first, stays first.

SORTING WITHOUT MODIFYING

sort modifies the original array. To sort without modifying:

Using slice:

```
let original = [3, 1, 4, 1, 5]

let sorted = original.slice().sort((a, b) => a - b)

console.log(original)
console.log(sorted)
```

Output:
[3, 1, 4, 1, 5]
[1, 1, 3, 4, 5]

Using spread:

let sorted = [...original].sort((a, b) => a - b)

Using toSorted (modern):

let sorted = original.toSorted((a, b) => a - b)

toSorted returns a new sorted array without modifying the original.

SORTING DATES

Sort date objects:

```
let dates = [
 new Date("2024-03-15"),
 new Date("2024-01-20"),
 new Date("2024-02-10")
]

dates.sort((a, b) => a - b)

console.log(dates.map(d => d.toISOString().split("T")[0]))
```

Output: ["2024-01-20", "2024-02-10", "2024-03-15"]

Date objects can be subtracted to get the difference in milliseconds.

Sort date strings:

```
let dateStrings = ["2024-03-15", "2024-01-20", "2024-02-10"]
```

dateStrings.sort()

console.log(dateStrings)

Output: ["2024-01-20", "2024-02-10", "2024-03-15"]

ISO date strings (YYYY-MM-DD) sort correctly as strings.

SORTING BOOLEANS

False before true:

```
let items = [
 { name: "A", active: true },
 { name: "B", active: false },
 { name: "C", active: true },
 { name: "D", active: false }
]
```

items.sort((a, b) => a.active - b.active)

console.log(items.map(i => `${i.name}: ${i.active}`))

Output: B: false, D: false, A: true, C: true

True before false:

items.sort((a, b) => b.active - a.active)

Boolean true is 1, false is 0 when used in math.

PRACTICAL EXAMPLES

Sort products by price:

```
function sortByPrice(products, ascending = true) {
 return [...products].sort((a, b) => {
 return ascending ? a.price - b.price : b.price - a.price
 })
}

let products = [
 { name: "A", price: 30 },
 { name: "B", price: 10 },
 { name: "C", price: 20 }
]

console.log(sortByPrice(products).map(p => p.name))
console.log(sortByPrice(products, false).map(p => p.name))
```

Output:
["B", "C", "A"]
["A", "C", "B"]

Sort by multiple criteria:

```
function sortUsers(users, criteria) {
 return [...users].sort((a, b) => {
 for (let criterion of criteria) {
 let { field, order } = criterion
 let comparison

 if (typeof a[field] === "string") {
 comparison = a[field].localeCompare(b[field])
 } else {
 comparison = a[field] - b[field]
 }

 if (order === "desc") {
 comparison = -comparison
 }

 if (comparison !== 0) {
 return comparison
```

```
 }
 }
 return 0
 })
}

let users = [
 { name: "Bob", age: 30, score: 85 },
 { name: "Alice", age: 25, score: 90 },
 { name: "Alice", age: 30, score: 88 }
]

let sorted = sortUsers(users, [
 { field: "name", order: "asc" },
 { field: "age", order: "desc" }
])

console.log(sorted.map(u => `${u.name}, ${u.age}`))
```

Output: ["Alice, 30", "Alice, 25", "Bob, 30"]

PRACTICE

Sort an array of numbers in ascending order.

Sort an array of objects by a string property.

Sort objects by one property, then by another.

Create a copy of an array, sort it, and leave the original unchanged.

Sort an array of dates.

WHAT YOU LEARNED

You learned that default sort is alphabetical by string.
You learned to use compare functions for custom ordering.
You learned (a, b) => a - b for ascending numbers.
You learned (a, b) => b - a for descending numbers.
You learned localeCompare for string sorting.
You learned to sort objects by properties.
You learned multi-level sorting.
You learned to sort without modifying using slice or toSorted.

The next chapter teaches you about multi-dimensional arrays.

BOOK 5: ARRAYS

Chapter 6: Multi-Dimensional Arrays

ARRAYS OF ARRAYS

A multi-dimensional array is an array that contains other arrays:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]
```

This is a 2D array (two-dimensional). Think of it as a grid with rows and columns.

ACCESSING ELEMENTS

Use two sets of brackets:

```
let matrix = [
 [1, 2, 3],
```

```
 [4, 5, 6],
 [7, 8, 9]
]

console.log(matrix[0])
console.log(matrix[0][0])
console.log(matrix[1][2])
console.log(matrix[2][1])
```

Output:
[1, 2, 3]
1
6
8

matrix[0] gives the first row.
matrix[0][0] gives the first element of the first row.
matrix[1][2] gives row 1, column 2 (value 6).

Think of it as matrix[row][column].

MODIFYING ELEMENTS

Assign with double brackets:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]

matrix[1][1] = 99

console.log(matrix)
```

Output:
[

```
 [1, 2, 3],
 [4, 99, 6],
 [7, 8, 9]
]
```

LOOPING THROUGH 2D ARRAYS

Use nested loops:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]

for (let row = 0; row < matrix.length; row++) {
 for (let col = 0; col < matrix[row].length; col++) {
 console.log(`[${row}][${col}] = ${matrix[row][col]}`)
 }
}
```

Output:

```
[0][0] = 1
[0][1] = 2
[0][2] = 3
[1][0] = 4
...
```

Using for...of:

```
for (let row of matrix) {
 for (let cell of row) {
 console.log(cell)
 }
}
```

Print as grid:

```
for (let row of matrix) {
 console.log(row.join("\t"))
}
```

Output:

```
1	2	3
4	5	6
7	8	9
```

CREATING 2D ARRAYS

Empty 2D array:

```
let grid = []

for (let i = 0; i < 3; i++) {
 grid.push([])
}

console.log(grid)
```

Output: [[], [], []]

Filled with values:

```
let rows = 3
let cols = 4
let grid = []

for (let i = 0; i < rows; i++) {
 let row = []
 for (let j = 0; j < cols; j++) {
 row.push(0)
 }
 grid.push(row)
}
```

```
console.log(grid)
```

Output:

```
[
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]
]
```

Using Array.from:

```
let grid = Array.from({ length: 3 }, () => Array(4).fill(0))

console.log(grid)
```

Same output.

JAGGED ARRAYS

Rows do not have to be the same length:

```
let jagged = [
 [1],
 [2, 3],
 [4, 5, 6],
 [7, 8, 9, 10]
]

for (let row of jagged) {
 console.log(row.join(", "))
}
```

Output:

```
1
2, 3
4, 5, 6
```

7, 8, 9, 10

COMMON OPERATIONS

Sum all elements:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]

let sum = 0

for (let row of matrix) {
 for (let cell of row) {
 sum += cell
 }
}

console.log(`Sum: ${sum}`)
```

Output: Sum: 45

Find maximum:

```
let max = matrix[0][0]

for (let row of matrix) {
 for (let cell of row) {
 if (cell > max) {
 max = cell
 }
 }
}

console.log(`Max: ${max}`)
```

Output: Max: 9

Sum each row:

```
let rowSums = matrix.map(row => row.reduce((sum, cell) => sum + cell, 0))

console.log(rowSums)
```

Output: [6, 15, 24]

Sum each column:

```
let numCols = matrix[0].length
let colSums = []

for (let col = 0; col < numCols; col++) {
 let sum = 0
 for (let row = 0; row < matrix.length; row++) {
 sum += matrix[row][col]
 }
 colSums.push(sum)
}

console.log(colSums)
```

Output: [12, 15, 18]

TRANSPOSE

Swap rows and columns:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6]
]
```

```
let transposed = []

for (let col = 0; col < matrix[0].length; col++) {
 let newRow = []
 for (let row = 0; row < matrix.length; row++) {
 newRow.push(matrix[row][col])
 }
 transposed.push(newRow)
}

console.log(transposed)
```

Output:

```
[
 [1, 4],
 [2, 5],
 [3, 6]
]
```

Original was 2x3. Transposed is 3x2.

PRACTICAL EXAMPLES

Tic-tac-toe board:

```
let board = [
 ["X", "O", "X"],
 [" ", "X", "O"],
 ["O", " ", " "]
]

function printBoard(board) {
 for (let row of board) {
 console.log(row.join(" | "))
 }
}
```

```
printBoard(board)
```

Output:
```
X | O | X
 | X | O
O |  |
```

Check for win:

```
function checkRows(board, player) {
 for (let row of board) {
 if (row.every(cell => cell === player)) {
 return true
 }
 }
 return false
}
```

Spreadsheet data:

```
let spreadsheet = [
 ["Name", "Age", "City"],
 ["Alice", 25, "New York"],
 ["Bob", 30, "Boston"],
 ["Charlie", 35, "Chicago"]
]

let headers = spreadsheet[0]
let data = spreadsheet.slice(1)

for (let row of data) {
 let person = {}
 for (let i = 0; i < headers.length; i++) {
 person[headers[i]] = row[i]
 }
 console.log(person)
}
```

Image pixels:

```
let image = [
 [[255, 0, 0], [0, 255, 0], [0, 0, 255]],
 [[255, 255, 0], [0, 255, 255], [255, 0, 255]],
 [[255, 255, 255], [0, 0, 0], [128, 128, 128]]
]

function getPixel(image, row, col) {
 let [r, g, b] = image[row][col]
 return { r, g, b }
}

console.log(getPixel(image, 0, 0))
```

Output: { r: 255, g: 0, b: 0 }

Game map:

```
let gameMap = [
 ["W", "W", "W", "W", "W"],
 ["W", ".", ".", ".", "W"],
 ["W", ".", "P", ".", "W"],
 ["W", ".", ".", "E", "W"],
 ["W", "W", "W", "W", "W"]
]

function findPlayer(map) {
 for (let row = 0; row < map.length; row++) {
 for (let col = 0; col < map[row].length; col++) {
 if (map[row][col] === "P") {
 return { row, col }
 }
 }
 }
 return null
}

console.log(findPlayer(gameMap))
```

Output: { row: 2, col: 2 }

3D ARRAYS

Three dimensions:

```
let cube = [
 [
 [1, 2],
 [3, 4]
 ],
 [
 [5, 6],
 [7, 8]
 ]
]
```

```
console.log(cube[0][1][0])
```

Output: 3

Access as cube[layer][row][column].

PRACTICE

Create a 3x3 matrix and print its diagonal.

Sum all elements in a 2D array.

Find the position of a specific value in a 2D array.

Transpose a matrix.

Create a 5x5 grid filled with X characters.

WHAT YOU LEARNED

You learned that 2D arrays are arrays of arrays.
You learned to access elements with double brackets.
You learned to loop with nested loops or nested for...of.
You learned to create empty and filled 2D arrays.
You learned about jagged arrays with varying row lengths.
You learned common operations: sum, max, transpose.
You learned practical uses: games, grids, spreadsheets.

The next chapter teaches you about destructuring and spread.

BOOK 5: ARRAYS
Chapter 7: Destructuring and Spread

ARRAY DESTRUCTURING

Destructuring extracts values from arrays into variables:

```
let colors = ["red", "green", "blue"]

let [first, second, third] = colors

console.log(first)
console.log(second)
console.log(third)
```

Output:
red
green
blue

Instead of:

```
let first = colors[0]
let second = colors[1]
let third = colors[2]
```

One line does it all.

SKIPPING ELEMENTS

Use commas to skip:

```
let numbers = [1, 2, 3, 4, 5]

let [first, , third] = numbers

console.log(first)
console.log(third)
```

Output:
1
3

The second element is skipped.

```
let [, , , fourth] = numbers

console.log(fourth)
```

Output: 4

Skip the first three.

REST PATTERN

Collect remaining elements with ...:

```
let numbers = [1, 2, 3, 4, 5]

let [first, second, ...rest] = numbers

console.log(first)
console.log(second)
console.log(rest)
```

Output:
1
2
[3, 4, 5]

The rest gets everything left over.

Must be last:

```
let [first, ...rest, last] = numbers
```

Error. Rest must be at the end.

DEFAULT VALUES

Provide defaults for missing elements:

```
let colors = ["red"]

let [first, second = "default"] = colors

console.log(first)
console.log(second)
```

Output:
red
default

The array only had one element. second gets the default.

SWAPPING VARIABLES

Classic way:

```
let a = 1
let b = 2

let temp = a
a = b
b = temp
```

Destructuring way:

```
let a = 1
let b = 2

[a, b] = [b, a]

console.log(a)
console.log(b)
```

Output:
2
1

No temporary variable needed.

NESTED DESTRUCTURING

Extract from nested arrays:

```
let nested = [1, [2, 3], 4]
```

```
let [first, [second, third], fourth] = nested

console.log(first)
console.log(second)
console.log(third)
console.log(fourth)
```

Output:
1
2
3
4

FUNCTION PARAMETERS

Destructure in function parameters:

```
function printCoords([x, y]) {
 console.log(`X: ${x}, Y: ${y}`)
}

printCoords([10, 20])
```

Output: X: 10, Y: 20

Return multiple values:

```
function minMax(numbers) {
 return [Math.min(...numbers), Math.max(...numbers)]
}

let [min, max] = minMax([5, 2, 8, 1, 9])

console.log(min)
console.log(max)
```

Output:
1
9

SPREAD OPERATOR

The spread operator (...) expands an array:

```
let numbers = [1, 2, 3]

console.log(...numbers)
console.log(1, 2, 3)
```

Output:
1 2 3
1 2 3

Both are the same. Spread unpacks the array.

COPYING ARRAYS

Create a shallow copy:

```
let original = [1, 2, 3]

let copy = [...original]

copy.push(4)

console.log(original)
console.log(copy)
```

Output:
[1, 2, 3]

[1, 2, 3, 4]

Modifying the copy does not affect the original.

COMBINING ARRAYS

Merge arrays:

```
let arr1 = [1, 2, 3]
let arr2 = [4, 5, 6]

let combined = [...arr1, ...arr2]

console.log(combined)
```

Output: [1, 2, 3, 4, 5, 6]

Add elements while combining:

```
let combined = [...arr1, 99, ...arr2, 100]

console.log(combined)
```

Output: [1, 2, 3, 99, 4, 5, 6, 100]

SPREAD IN FUNCTION CALLS

Pass array elements as arguments:

```
function sum(a, b, c) {
 return a + b + c
}

let numbers = [1, 2, 3]
```

```
console.log(sum(...numbers))
```

Output: 6

Same as sum(1, 2, 3).

With Math functions:

```
let numbers = [5, 2, 8, 1, 9]

console.log(Math.max(...numbers))
console.log(Math.min(...numbers))
```

Output:
9
1

SPREAD WITH STRINGS

Spread a string into characters:

```
let word = "hello"

let letters = [...word]

console.log(letters)
```

Output: ["h", "e", "l", "l", "o"]

PRACTICAL EXAMPLES

Extract first and rest:

```
let items = ["first", "second", "third", "fourth"]

let [head, ...tail] = items

console.log(head)
console.log(tail)
```

Output:
first
["second", "third", "fourth"]

Remove duplicates:

```
let numbers = [1, 2, 2, 3, 3, 3, 4]

let unique = [...new Set(numbers)]

console.log(unique)
```

Output: [1, 2, 3, 4]

Clone and modify:

```
let original = [1, 2, 3]

let modified = [...original, 4, 5]

console.log(modified)
```

Prepend items:

```
let existing = [4, 5, 6]

let withPrefix = [1, 2, 3, ...existing]

console.log(withPrefix)
```

Output: [1, 2, 3, 4, 5, 6]

Insert in middle:

```
let arr = [1, 2, 5, 6]

let withInsert = [...arr.slice(0, 2), 3, 4, ...arr.slice(2)]

console.log(withInsert)
```

Output: [1, 2, 3, 4, 5, 6]

Parse coordinates:

```
function drawLine([x1, y1], [x2, y2]) {
 console.log(`Line from (${x1}, ${y1}) to (${x2}, ${y2})`)
}

drawLine([0, 0], [100, 100])
```

Output: Line from (0, 0) to (100, 100)

RGB color handling:

```
let rgb = [255, 128, 64]

let [r, g, b] = rgb

console.log(`R: ${r}, G: ${g}, B: ${b}`)

let brighter = rgb.map(c => Math.min(255, c + 50))

console.log(brighter)
```

DESTRUCTURING VS SPREAD

Destructuring: Extract values INTO variables
Spread: Expand values OUT OF an array

Destructuring:

let [a, b, c] = [1, 2, 3]

Values go into a, b, c.

Spread:

let arr = [1, 2, 3]
let copy = [...arr]

Values come out of arr into copy.

Rest in destructuring (looks like spread):

let [first, ...rest] = [1, 2, 3, 4, 5]

Collects remaining values into rest array.

PRACTICE

Destructure the first two elements of an array.

Swap two variables using destructuring.

Copy an array using spread.

Combine three arrays into one.

Extract first element and collect the rest.

WHAT YOU LEARNED

You learned destructuring extracts array values into variables.
You learned to skip elements with commas.

You learned the rest pattern (...) collects remaining elements.
You learned default values for missing elements.
You learned spread expands arrays.
You learned to copy and combine arrays with spread.
You learned to pass arrays as function arguments with spread.

The next chapter brings everything together with practice programs.

BOOK 5: ARRAYS

Chapter 8: Practice Array Programs

INVENTORY MANAGEMENT SYSTEM

Build a product inventory tracker:

```
let inventory = [
 { id: 1, name: "Laptop", category: "Electronics", price: 999, quantity: 15 },
 { id: 2, name: "Phone", category: "Electronics", price: 699, quantity: 30 },
 { id: 3, name: "Shirt", category: "Clothing", price: 29, quantity: 100 },
 { id: 4, name: "Pants", category: "Clothing", price: 49, quantity: 75 },
 { id: 5, name: "Headphones", category: "Electronics", price: 199, quantity: 50 }
]

function getInventoryValue() {
 return inventory.reduce((total, item) => {
 return total + (item.price * item.quantity)
 }, 0)
}

function getByCategory(category) {
 return inventory.filter(item => item.category === category)
```

```
}

function getLowStock(threshold) {
 return inventory.filter(item => item.quantity < threshold)
}

function findProduct(id) {
 return inventory.find(item => item.id === id)
}

function updateQuantity(id, change) {
 let product = findProduct(id)
 if (product) {
 product.quantity += change
 return true
 }
 return false
}

function getSortedByPrice(ascending = true) {
 return [...inventory].sort((a, b) => {
 return ascending ? a.price - b.price : b.price - a.price
 })
}

console.log("=== INVENTORY SYSTEM ===")
console.log(`Total Value: $${getInventoryValue()}`)
console.log("")
console.log("Electronics:")
getByCategory("Electronics").forEach(item => {
 console.log(`  ${item.name}: $${item.price} (${item.quantity} in
stock)`)
})
console.log("")
console.log("Low Stock (< 20):")
getLowStock(20).forEach(item => {
 console.log(`  ${item.name}: ${item.quantity} units`)
})
console.log("")
console.log("Sorted by price:")
```

```
getSortedByPrice().forEach(item => {
 console.log(`  ${item.name}: $${item.price}`)
})
```

DATA ANALYSIS TOOLKIT

Statistical functions for arrays:

```
function sum(arr) {
 return arr.reduce((a, b) => a + b, 0)
}

function mean(arr) {
 return arr.length ? sum(arr) / arr.length : 0
}

function median(arr) {
 if (!arr.length) return 0
 let sorted = [...arr].sort((a, b) => a - b)
 let mid = Math.floor(sorted.length / 2)
 return sorted.length % 2 ? sorted[mid] : (sorted[mid - 1] +
sorted[mid]) / 2
}

function mode(arr) {
 let counts = arr.reduce((acc, val) => {
 acc[val] = (acc[val] || 0) + 1
 return acc
 }, {})

 let maxCount = Math.max(...Object.values(counts))
 let modes = Object.keys(counts)
 .filter(key => counts[key] === maxCount)
 .map(Number)

 return modes
}
```

```
function range(arr) {
 return Math.max(...arr) - Math.min(...arr)
}

function variance(arr) {
 let avg = mean(arr)
 let squaredDiffs = arr.map(x => Math.pow(x - avg, 2))
 return mean(squaredDiffs)
}

function standardDeviation(arr) {
 return Math.sqrt(variance(arr))
}

function percentile(arr, p) {
 let sorted = [...arr].sort((a, b) => a - b)
 let index = Math.ceil((p / 100) * sorted.length) - 1
 return sorted[Math.max(0, index)]
}

let scores = [85, 92, 78, 90, 88, 76, 95, 82, 79, 91]

console.log("=== DATA ANALYSIS ===")
console.log(`Data: ${scores.join(", ")}`)
console.log(`Sum: ${sum(scores)}`)
console.log(`Mean: ${mean(scores).toFixed(2)}`)
console.log(`Median: ${median(scores)}`)
console.log(`Mode: ${mode(scores)}`)
console.log(`Range: ${range(scores)}`)
console.log(`Std Dev: ${standardDeviation(scores).toFixed(2)}`)
console.log(`25th Percentile: ${percentile(scores, 25)}`)
console.log(`75th Percentile: ${percentile(scores, 75)}`)
```

TASK SCHEDULER

A priority-based task queue:

```
let tasks = []
let nextId = 1

function addTask(name, priority = 1) {
 tasks.push({
 id: nextId++,
 name,
 priority,
 createdAt: new Date(),
 completed: false
 })
 sortByPriority()
 return tasks[tasks.length - 1]
}

function sortByPriority() {
 tasks.sort((a, b) => b.priority - a.priority)
}

function getNextTask() {
 return tasks.find(t => !t.completed)
}

function completeTask(id) {
 let task = tasks.find(t => t.id === id)
 if (task) {
 task.completed = true
 task.completedAt = new Date()
 return true
 }
 return false
}

function getPendingTasks() {
 return tasks.filter(t => !t.completed)
}

function getCompletedTasks() {
 return tasks.filter(t => t.completed)
```

```
}

function getTasksByPriority(priority) {
 return tasks.filter(t => t.priority === priority)
}

addTask("Critical bug fix", 5)
addTask("Update documentation", 1)
addTask("New feature", 3)
addTask("Security patch", 5)
addTask("Code review", 2)

console.log("=== TASK SCHEDULER ===")
console.log("All tasks by priority:")
tasks.forEach(t => {
 console.log(`  [${t.priority}] ${t.name}`)
})

console.log("")
console.log(`Next task: ${getNextTask().name}`)

completeTask(1)
completeTask(4)

console.log("")
console.log("Pending tasks:")
getPendingTasks().forEach(t => {
 console.log(`  ${t.name}`)
})

console.log("")
console.log("Completed tasks:")
getCompletedTasks().forEach(t => {
 console.log(`  ${t.name}`)
})
```

PLAYLIST MANAGER

Music playlist with shuffle and history:

```
let playlist = [
 { id: 1, title: "Song A", artist: "Artist 1", duration: 180 },
 { id: 2, title: "Song B", artist: "Artist 2", duration: 240 },
 { id: 3, title: "Song C", artist: "Artist 1", duration: 200 },
 { id: 4, title: "Song D", artist: "Artist 3", duration: 300 },
 { id: 5, title: "Song E", artist: "Artist 2", duration: 220 }
]

let currentIndex = 0
let playHistory = []
let shuffleMode = false

function formatDuration(seconds) {
 let mins = Math.floor(seconds / 60)
 let secs = seconds % 60
 return `${mins}:${secs.toString().padStart(2, "0")}`
}

function getTotalDuration() {
 return playlist.reduce((total, song) => total + song.duration, 0)
}

function getCurrentSong() {
 return playlist[currentIndex]
}

function nextSong() {
 playHistory.push(currentIndex)
 if (shuffleMode) {
 let available = playlist
 .map((_, i) => i)
 .filter(i => i !== currentIndex)
 currentIndex = available[Math.floor(Math.random() *
available.length)]
 } else {
 currentIndex = (currentIndex + 1) % playlist.length
 }
```

```
 return getCurrentSong()
}

function previousSong() {
 if (playHistory.length > 0) {
 currentIndex = playHistory.pop()
 }
 return getCurrentSong()
}

function shuffle() {
 let shuffled = [...playlist]
 for (let i = shuffled.length - 1; i > 0; i--) {
 let j = Math.floor(Math.random() * (i + 1))
 ;[shuffled[i], shuffled[j]] = [shuffled[j], shuffled[i]]
 }
 playlist = shuffled
 currentIndex = 0
 playHistory = []
}

function getByArtist(artist) {
 return playlist.filter(song => song.artist === artist)
}

console.log("=== PLAYLIST MANAGER ===")
console.log(`Total songs: ${playlist.length}`)
console.log(`Total duration: ${formatDuration(getTotalDuration())}
`)
console.log("")
console.log("Playlist:")
playlist.forEach((song, i) => {
 let marker = i === currentIndex ? ">" : " "
 console.log(`${marker} ${song.title} - ${song.artist}
(${formatDuration(song.duration)})`)
})
console.log("")
console.log(`Now playing: ${getCurrentSong().title}`)
nextSong()
console.log(`Next: ${getCurrentSong().title}`)
```

```
nextSong()
console.log(`Next: ${getCurrentSong().title}`)
previousSong()
console.log(`Previous: ${getCurrentSong().title}`)
```

MATRIX CALCULATOR

Operations on 2D matrices:

```
function createMatrix(rows, cols, fill = 0) {
 return Array.from({ length: rows }, () => Array(cols).fill(fill))
}

function addMatrices(a, b) {
 return a.map((row, i) => row.map((val, j) => val + b[i][j]))
}

function subtractMatrices(a, b) {
 return a.map((row, i) => row.map((val, j) => val - b[i][j]))
}

function scalarMultiply(matrix, scalar) {
 return matrix.map(row => row.map(val => val * scalar))
}

function transpose(matrix) {
 return matrix[0].map((_, colIndex) =>
 matrix.map(row => row[colIndex])
 )
}

function multiplyMatrices(a, b) {
 let result = createMatrix(a.length, b[0].length)

 for (let i = 0; i < a.length; i++) {
 for (let j = 0; j < b[0].length; j++) {
 for (let k = 0; k < b.length; k++) {
```

```
 result[i][j] += a[i][k] * b[k][j]
 }
 }
 }

 return result
}

function printMatrix(matrix, name = "Matrix") {
 console.log(`${name}:`)
 matrix.forEach(row => {
 console.log("  " + row.map(v => v.toString().padStart(4)).join("
"))
 })
}

let A = [
 [1, 2],
 [3, 4]
]

let B = [
 [5, 6],
 [7, 8]
]

console.log("=== MATRIX CALCULATOR ===")
printMatrix(A, "Matrix A")
printMatrix(B, "Matrix B")
console.log("")
printMatrix(addMatrices(A, B), "A + B")
printMatrix(subtractMatrices(A, B), "A - B")
printMatrix(scalarMultiply(A, 2), "A * 2")
printMatrix(transpose(A), "Transpose of A")
printMatrix(multiplyMatrices(A, B), "A * B")
```

SHOPPING CART SYSTEM

Complete e-commerce cart:

```
let cart = []
let products = [
 { id: 1, name: "Laptop", price: 999.99 },
 { id: 2, name: "Mouse", price: 29.99 },
 { id: 3, name: "Keyboard", price: 79.99 },
 { id: 4, name: "Monitor", price: 299.99 }
]

function addToCart(productId, quantity = 1) {
 let product = products.find(p => p.id === productId)
 if (!product) return false

 let existing = cart.find(item => item.productId === productId)
 if (existing) {
 existing.quantity += quantity
 } else {
 cart.push({
 productId,
 name: product.name,
 price: product.price,
 quantity
 })
 }
 return true
}

function removeFromCart(productId) {
 let index = cart.findIndex(item => item.productId ===
productId)
 if (index !== -1) {
 cart.splice(index, 1)
 return true
 }
 return false
}

function updateQuantity(productId, quantity) {
```

```
 let item = cart.find(item => item.productId === productId)
 if (item) {
 item.quantity = quantity
 if (item.quantity <= 0) {
 removeFromCart(productId)
 }
 return true
 }
 return false
}

function getSubtotal() {
 return cart.reduce((total, item) => total + (item.price *
item.quantity), 0)
}

function getItemCount() {
 return cart.reduce((count, item) => count + item.quantity, 0)
}

function applyDiscount(percentage) {
 let subtotal = getSubtotal()
 return subtotal * (1 - percentage / 100)
}

function printCart() {
 console.log("Shopping Cart:")
 if (cart.length === 0) {
 console.log("  (empty)")
 return
 }

 cart.forEach(item => {
 let lineTotal = item.price * item.quantity
 console.log(`  ${item.name} x${item.quantity} @ $${item.price}
= $${lineTotal.toFixed(2)}`)
 })

 console.log(`  ---`)
 console.log(`  Subtotal: $${getSubtotal().toFixed(2)}`)
```

```
 console.log(`  Items: ${getItemCount()}`)
}

addToCart(1, 1)
addToCart(2, 2)
addToCart(3, 1)
addToCart(2, 1)

console.log("=== SHOPPING CART ===")
printCart()
console.log("")
console.log(`With 10% discount: $${applyDiscount(10).toFixed(2)}
`)
```

WHAT YOU BUILT

You built six complete programs:

Inventory management with search, filter, and sort
Data analysis with statistics functions
Priority task scheduler
Music playlist with navigation
Matrix calculator with operations
Shopping cart system

Each program combined:

Array creation and manipulation
Search methods (find, filter, includes)
Transformation methods (map, reduce)
Sorting with custom comparators
Destructuring and spread
Nested arrays and objects

WHAT COMES NEXT

You now have strong foundations:

- Variables store data (Book 1)
- Decisions control flow (Book 2)
- Loops process collections (Book 3)
- Functions organize code (Book 4)
- Arrays hold collections (Book 5)

But programs need structure. Real applications organize code into objects that represent things in the world. A user has properties and can do things. A product has properties and behaviors.

Book 6 teaches you objects in depth. You will learn:

- Creating and using objects
- Object properties and methods
- Object-oriented programming concepts
- Working with JSON data
- Object manipulation patterns

Objects are how JavaScript organizes complex data. Combined with arrays, they let you model anything.

You finished Book 5. You can work with collections. Now we model the real world.